Danny Hillis, Notes, 7/10-11/86

These are some notes on a possible architecture for the next machine. This is an attempt to pull together some of the architectural lessons that we have been learning from the current machines and to bring together some of the good architectural ideas that have generated internally into some comprehensible form. In particular, Alex, Bradley, Brewster, Dick, Guy, and Rolf will notice some of their favorite architectural ideas incorporated into this proposal.

The purpose of this machine, as I see it, is threefold:

- First, we would like to do a better job of what we are currently doing well. In particular, the best ideas like data level parallelism, general communication, and virtual processors should be implemented as well as we feel they can be. This means that we have to be more aggressive on the technology.

- Second, we need to do a better job of things that we currently do not do well. In particular, we need to do a better job on executing code that has a large sequential component or large component that is only slightly parallel. We should also be able to implement some kind of control level parallelism because by the time the machine comes out people will probably be understanding how to program it.

- Third, I believe that we should be aiming toward making an arbitrarily expandable multi-tasking machine that is capable of handling large databases. I think that this means it must be fault tolerant.

This is a very different set of requirements than our original conception of the Connection Machine, but I think that our initial experiences with marketing the machine, in programming it, and in using it ourselves point strongly in this direction.

With these goals in mind, the following are notes for our next DARPA proposal. Like our last DARPA proposal, this is not intended to be a detailed specification of the machine that we will actually build; but rather a specific paper tiger example of the kind of machine that we would want to build. We will have plenty of time to go back and redesign it later.

Kinds of Parallelism

There are many kinds of parallelism in modern computers. The simplest type of parallelism and, the most common, is *word parallelism*, that is, the parallelism inherent within a single arithmetic operation among multiple bit numbers. Depending on the type of operation performed, this typically ranges somewhere between 1 and 64 bits.

The second most common form of parallelism is *functional parallelism*, where different components of a single task can be executed more or less independently with the output of one going into the next. For example, some computers use separate processors for handling input/output channels. At a lower level, many computers implement functional parallelism by executing one instruction while fetching the next. In addition, many large grain parallel computers allow a more general form of functional parallelism to be explicitly specified by the program. For example, in a robotics calculation one processor may be programmed to control each joint in a robot arm. Some compilers are able to automatically extract functional parallelism from normal serial code by noticing operations that can be executed simultaneously. And some computers such as very long instruction word machines have multiple functional units to take advantage of this type of compilation. Typically parallelism of this sort is somewhere between a factor of 10 and 100.

Another type of parallelism that is particularly simple is *multi-tasking*. This is the type of parallelism that is being exploited when a time-sharing system is replaced by a loosely coupled network of personal computers or workstations. Some parallel processors are primarily designed to take advantage of this type of multi-tasking parallelism; for example, to support multiple users simultaneously or multiple, more or less independent tasks per user. It is not uncommon for a database system to have hundreds or even thousands of users simultaneously.

A final form of parallelism which offers the greatest potential concurrency is *data parallelism*, in which all of the data associated with a given application is operated upon simultaneously. Vector processing is a very simple form of data parallelism. Potential data parallelism also exists in tasks like updating all elements in a physical simulation, looking at all points in an image, or simultaneously searching all the documents in a database. Problems which exhibit a data parallelism of the order of millions or even hundreds of millions are not uncommon.

Table 1: Comparison of Types of Parallelism

Type of Parallelism	*Examples*	*Typical Values*
Word Parallelism	Multiple bit addition	1-64
Functional Parallelism	Separate processor for i/o channels	10-100
Multi-Task Parallelism	A loosely coupled network of workstations	1-1,000
Data Parallelism	Simultaneously searching all documents of a database	1-1,000,000

Because data parallelism has much larger potential gain than any of the other forms, the Connection Machine system was specifically designed to exploit this type of parallelism. It has no hardware to take advantage of task, functional, or word parallelism. In this sense, the Connection Machine is a very pure and simple design; but it is by no means optimal. A machine which was able to exploit many kinds of parallelisms would obviously outperform a purely data parallel computer. In the best case, it would actually be possible to achieve multiplicative gains of the various kinds of parallelism. For example, a word parallel Connection Machine would achieve a total parallelism equal to the product of the average number of bits per word and the average number of data elements being operated on simultaneously. The CM2 is designed to exploit all of the forms of parallelism listed above. In particular, multi-tasking which, next to data parallelism, offers the highest potential gain.

The CM2

Figure 1 shows schematically how the Connection Machine looks to a programmer. A single task operates on sets of data simultaneously corresponding to virtual processors. In the physical implementation of the Connection

Machine a single task is embodied in the host processor and groups of virtual processes are embedded in the physical processing elements. The communications unit allows the data to interact so that it may be processed in parallel. Since the physical processors operate bit serially, there is no word parallelism and since they are operating from a single instruction stream between the host and the physical processors, there is no general form of functional parallelism. Also since there is only one uni-processor host, there is no hardware and body meta-task level parallelism.

Figure 3 shows a similar abstract model of the CM2. Here we have essentially two sorts of virtual procesors: *virtual data processors*, which operate multiple data elements simultaneously, and *virtual control processors*, which correspond to multiple tasks or multiple functions within the same task. In the case of essentially independent tasks, the virtual control processors talk to essentially independent sets of data processors; whereas in the case of a functional decomposition of a single task, the virtual control processors are matched through access to common data elements. The hardware which supports this model is shown schematically in Figure 4.

Notice that, unlike the CM1, the CM2 has no host processor. Users connect into the machine through network connections or potentially directly to data terminal devices (see Figure 5). There are no standard sequential computers in the system, although a physical control processor and a physical data processor can form essentially the data path of a sequential computer. Figure 5 shows how the CM1 is connected from the host to the user and how the CM2 is connected through the network to multiple users.

Physical vs. Virtual Architectures

Any computer architecture represents a compromise between two fundamentally conflicting sets of requirements. On one hand, an architecture should be entirely technology independent with the form and function of its various units arranged elegantly, to suit the needs of the programmer. In conflict with this is the need to implement the architecture. A good implementation should be able to take advantage of the particular properties and capabilities of the components. One way to resolve this conflict is to present the programmer with a *virtual architecture* which is designed according to the needs of the programmer, that is implemented by a *physical* *architecture* designed to best take advantage of the available technology.

An example of such an abstraction is the memory system of many computers today. The programmer sees a machine with a simple uniform address space, but this may be implemented by a hierarchy of fast cache memory, conventional random access memory, and secondary storage. This is an example of a separation between the virtual and the physical architecture. The virtual architecture presents a single uniform address space and the physical architecture implements it through a hierarchical system of storage.

In the CM1, the physical and the virtual architecture are very close to one another. At the physical level the architecture has 64,000 bit serial processors, each with 4000 bits of memory. At the virtual architecture level the CM1 has some integer multiple at 64,000 virtual processors, each executing a variable length instruction set. In the simplest cases, the physical and the virtual machine are essential identical. In the CM2, on the other hand, while the virtual machine is very similar to the CM1, the physical machine is very different. For this reason, the virtual and physical architectures of the CM2 will be described separately.

The Meta-Machine, The Virtual Architecture of the CM2

The essential good idea in the Connection Machine architecture is that each unit of data should have its own processor and that the processors should be able to communicate. A weakness of the Connection Machine is that each piece of data is given only a von Neumann processor. Each piece of data is given a Connection Machine, or more precisely, a Meta-Machine, recursively. The data parallelism is available recursively, to match the recursive structure of data.

The basic operations of the Meta-Machine are essentially the same as those of the Connection Machine, except that they can be specified more elegantly since the possible operations of the data level cells are exactly the same as the possible operations of the whole machine. A processor in the Meta-Machine has the following capabilities:

1. It can execute normal von Neumann type operations including arithmetic logic operations, data movement, and normal control flow of operations such as subroutine calls and branches.

2. It can create a copy of a portion of the database and assign a processor to each copy of the item. This operation is called *consing* and it involves allocating new processors. The allocating processor is called the *control processor* and the allocated processors are called *data processors*. These are relative terms since the data processors have the full capabilities of the control processors and are able to allocate data processors of their own.

3. A processor is able to select a *context set* from among its allocated data processors. This context set is the set of data to be operated upon in parallel. The context set is chosen according to some condition applied to all of the data processors or to all of the data processors in the current context set. Context sets may be saved and restored.

4. A processor may perform parallel operations concurrently on all of the data in its context set. The parallel operations are exactly the same as the sequential operations in category 1, except that they are applied to all data in the context set concurrently. These include all data manipulations, memory referencing (communications), and control flow operations. As far as the programmer is able to see, these operations take place *simultaneously* on all processors in the data set. (The distinction between *concurrently* and *simultaneously* is only important for the instructions that involve communications between processors.)

5. The processor is able to update the database according to the changes computed by its data processors, and to deallocate data processors.

The Physical Architecture

The physical architecture of the CM2 is shown schematically in Figure 6. The control store on the left is essentially the program storage for all tasks. This store is conceptually accessible simultaneously by any or all of the control processors and it is where programs reside. A single program may be shared by multiple tasks or users. The network between the control store and the physical control processors is designed in such a way that multiple access to the same section of program can be handled efficiently. This program memory may be loaded directly from the data memory, although that path is not shown on the diagram. The physical control processors implement instruction fetching and interpretation and are responsible for executing control flow instructions such as calls and jump operations, and for orchestrating the behavior of sets of physical data processors. The physical control processors and physical data processors are connected by a network which efficiently allows the broadcast of data execution instructions to multiple physical data processors simultaneously. In addition, it allows each physical control processor to access individual items of data through the physical data processors into individual items of data memory through the physical control processors, much like the operation of the address decoding logic in a conventional memory. In addition, it allows information to be combined from a group of processors by a simple inclusive-or operation. The physical data processors is where the actual operations on the data are performed. Typical operations include normal word parallel operations such as arithmetic field extraction and bitwise boolean functions. When floating point extension hardware is included these units also support full precision floating point arithmetic. The physical data processors are connected to the data memory through a communications network that is very similar to the routers of the Connection Machine. It allows direct memory access with a full memory, including combining functions such as fetch and add. As in the Connection Machine, the memory accessing scheme is designed to exploit locality in that each physical data processor has a portion of mmemory that it can access at much higher rates than the rest of the memory. Each physical data processor has its own portion of memory which it can access at much higher rates than other portions of memory. The other networks in the CM2, such as the network between the physical control processors and the physical data processors are also designed to exploit locality in a similar manner.

The CM2 supports a virtual memory architecture that goes beyond the simple mapping of virtual data processors and the physical data processors. It also supports virtual memory in the traditional sense. Both the control memory and the data memory can be demand paged from the disk unit. As with the rest of the Connection Machine, the disks operate independently and in parallel under the direct control of the physical control processors. The data memory is dual ported in such a way so that one portion may be loaded from disks while another portion is being accessed by the physical data processors. Input/output connections to the network are display devices (display devices are video cameras) and is directly mapped into data memory in a similar manner.

One advantage that the CM2 has over the conventional Connection Machine is that it is able to execute even sequential (non-parallel) segments of code for efficiency. This is important because since most programs have at least some portion of sequential code and since there is no first processor to execute this portion of the code, unless the Connection Machine executed this code efficiently, it would dominate the time required for computation. The Connection Machine's efficiency in executing sequential code small amounts of parallelism, comes from essentially two sources. First, since there is a tight coupling between a physical control processor and its local memory, a pair of these form a high performance (10 MIPS) serial machine. Second, since the Connection Machine efficiently supports multitasking, even while one portion of the machine is executing simple serial code, the rest of the machine can be utilized efficiently.

Instruction Set

The instructions of the CM2 bear a close relationship to the instructions of a conventional machine. They are divided into three categories:

- local instructions;
- parallel instructions; and
- context instructions.

The local instructions are exactly the instructions of a conventional machine, including subroutine calls, conditional and unconditional branches, returns, register-based arithmetic data movement, logical operations, and testing. The local instructions are executed within the control processor. The parallel data instructions are exactly like the local data instructions except that they are executed on a set of parallel data processors. This set is called the *context set*. Groups of these instructions called orders are executed on all virtual data processors in the context set simultaneously. See section on orders. For each local data instruction there is a corresponding parallel data instruction.

The context instructions are used to specify the set of virtual data processors to be executed upon in parallel. There are four context instructions:

1. Set the context to be all virtual processors satisfying some condition.
2. Restrict the context to be some subcontext of processors within the current context satisfying some condition.
3. Push the current context onto a stack.
4. Pop the current context off the stack.

These context instructions may be intermixed with parallel data instructions into groups to form orders.

While we have not yet selected exactly which instruction set to use, we will probably select an instruction set corresponding directly to the instruction set of some currently available microprocessor or minicomputer; that is, the control and local data instruction sets will be exactly the instruction sets of the corresponding conventional computer. The parallel instruction set will correspond to the local instruction set on a one-to-one basis. The context instructions will be as specified above. Possible candidates include the VAX instruction set, IBM 360 instruction set, the 68000, the 80386, and various other machines. The use of a proprietary instruction set would, of course, require the obtainment of proper licenses.

One attractive implementation possibility for the CM2 is the use of conventional microprocessors in an unconventional way. Specifically, we could use a custom chip in between the microprocessor and the memory which serves as a communications network and serves to intercept the parallel versions of the instructions and extended constructions involving things like context switching. This chip essentially serves as a memory management unit with local caching for both instructions and data; in particular, data read over the communications network. This is attractive from an implementation standpoint because it allows us to concentrate our resources on those parts of the implementation that are unique to our architecture, specifically the memory management, instruction distribution, and communications. Since there are several types of communications that take place, they will each be addressed separately, although they may be handled physically by only one or a few networks.

- Local Memory Data Referencing: For local references to memory the custom chip will need to solve exactly the same problem as a conventional memory management unit, presuming that we use conventional methods. For example, a cache followed by a page table which references into physical random access memory or creates page faults onto disks. We will probably want to use a microprocessor that is designed to support these functions with an external memory management unit. Since we will probably be using a single physical processor to simulate many virtual processors, we may want to put some form of virtual processor support into the memory management unit (see section on Context Lists).

- Global Memory Read: References to the memory of other virtual processors in general go to the router, although it may be desirable to cache references to memory that is not to change. This cache may be stored within the local cache of the custom chip, or in the random access memory associated with the chip. (Or conceivably even on local secondary storage.)

- Global Memory Write: These are operations that correspond to the beta operations on the current Connection Machine. They include things like send with at, send with overwrite, and also things like what is currently called global-or. Notice that these operations imply some form of synchronization, at least in some cases (see section on Synchronization). Operations like global-or or synchronized global-write from the data processors to the control processor.

- Sequential Instruction Fetching: These are the normal instruction fetches of the machine. The issues here are very similar to those in local and global memory referencing, except, of course, since the data is read only, caching is even more desirable. Again, two levels of caching are probably appropriate; one in fast storage within the custom chip, another in local random access memory or secondary storage.

- Parallel Instruction Distribution: This is the broadcast mode distribution of instructions from a control processor to the local data processors. Since these instructions must be distributed to each virtual data processor within the physical processor, they will probably be cached in some form of local memory, where they can be called as a subroutine by each virtual data processor where they will be executed as sequential instructions. Multiple levels of parallelism would be executed by a parallel call instruction which, although executed as a sequential call instruction by each data processor, could perform a subroutine called a parallel code.

- Allocation Communication: The communication involved in the allocation of data processors to control processors is unique in that it is not restricted to the current context. In the worst case, it may be necessary to resort to a global broadcast, but since the system is intended to be arbitrarily expandable, this cannot be the usual case. Presumably this needs to be implemented by some form of local directory storage or some spreading search wave so that processors can be allocated without tying up the entire machine.

Virtual Data Processors

Each physical data processor is expected to simulate a fairly large number of virtual data processors. To allow the efficient simulation of virtual processors are very different sizes and also to efficiently take advantage of the fact that many virutal processors are not within a given context set, the CM2 uses a different representation of virtual processors. Since this representation is critical to the efficiency of the machine, it may be worthwhile to specialize the hardware to handle it efficiently. Each physical data processor keeps a looped list of pointers to the sections of memory containing the virtual data processors for a given task. This list is resorted each time the context is restricted so that the processor can keep the context set by a series of successive pointers into nested context. Executing an order for all the virtual processors consists of cdring down this length list, starting from the innermost context point. For each processor the parallel instructions are executed from the order cache, relative to the start address of the current virtual data processor. Thus, the number of times the order is executed depends only on the number of virtual processors that are turned on in a given context.

Besides handling a set of virtual processors for a given control processor, the physical data processor may also be handling multiple context for multiple tasks. These are handled in sequence by a similar mechanism as the virtual data processors within a given context. The physical data processor keeps a link list of the contexts for each task contained within a linked list of tasks.

If the orders are small the virtual processing mechanism may incur a significant amount of overhead if it is done purely under program control. It may be worth building in the linked list representation of the virtual processors into the look-ahead mechanism for the cache so that this process may be handled efficiently. It is also possible to put it entirely within hardware, even if conventional microprocessors are used as the physical data processor.

Orders

The unit of communication between a control processor and a data processor is called an *order*. In the simplest case, an order is a single instruction. An order may also be a group of instructions which may be executed together without concern for synchronization across physical data processors within the order. In other words, the order is the minimum unit of synchronization in the machine. The basic action of a physical control processor is to issue an order, wait for confirmation that it has been executed by all virtual data processors, and then issue another order. Different virtual processors can and in general will execute various instructions within an order at different times.

It is the job of the order network to broadcast orders from physical control processors to physical data processors and to signal the physical control processors when an order has been executed by all processors. This signalling mechanism is also used to combine condition codes for control of programming flow and to signal errors. In this sense, the order network replaces the instruction broadcast mechanism and the global-or tree of the CM1.

Because the order network is relatively low bandwidth compared to the data network, information on this network is transmitted in serial. The broadcast order from the control unit are broadcast to all physical data processors where they are filtered and orders corresponding to relevant tasks are stored in the order caches corresponding to each physical data processor. Orders contain no control flow instructions, only data operation directions.

Each order cache stores the task IDs of all tasks that are currently being executed by its corresponding physical data processor. Each order is issued with a task tag and the order cache will only store orders whose tags correspond to the task being processed.

Fault Tolerance

One of the primary design goals of the CM2 is 100% up-time. This is achieved through a combination of of transaction-based software protocols and dynamically reconfigurable self-checking hardware. From a user point of view, the system is always up, although particular pieces of the system may be nonfunctional at any given time.

The user's interactions with the system are in terms of *transactions* on a *database*. The system guarantees that it will always be able to process transactions at a certain guaranteed rate and that the integrity of the database will always be maintained even if hardware failures occur during the course of a transaction. If such a failure has occurred, the system is allowed to abort a transaction and then reprocess it, so if a failure occurs during the course of a transaction it may take longer than a normal transaction, but it will be computed correctly. The database modifications specified by a transaction do not appear in the shared database until the entire transaction is completed.

The system works as follows. When a transaction request comes in, generally from the network, a control processor assigns the incoming address with a unique systemwide task identification number. The request will not be cleared from memory until the transaction is actually completed, but the task ID associated with it will serve to show the other control processors that it is already being processed.

Each item in the database is stored redundantly on two physically separated mass storage units. One copy is the primary copy, the other is a back-up. To process a transaction, a control processor establishes connections through the instruction network to the physical data processors corresponding in control of the primary copies of the relevant items in the database. It then proceeds to process the transaction, creating whatever temporary structures are necessary, but not actually updating the database, but merely recording the changes to be made in locala copies of the modified section of the database. This local partial copy is called the *journal.* If an uncorrectable error in one of the data processors is detected during this phase of the operation, then the control processor will reprocess the transaction from the beginning.

Once the transaction is completed, the control processor locks the relevant portions of the database against access and updates the shared database from the journal. If one of the data processors goes off-line during this portion of the transaction, then the relevant portions of the primary database will be reconstructed from the secondary copy and the transaction will be restarted. Notice that in this case the primary databases of all of the data processors involved in the transaction must be reconstructed, not just the data processor that went off-line. After this phase has been successfully completed, the updates are shipped to the physical data processors corresponding to the secondary copies of the database. If an error occurs during this phase of the operation, then the entire relevant portion of the secondary copy of the database is reconstructed from the primary copy. Once the updates has been performed on the secondary copy of the database, then the transaction is completed and the requests may be cleared.

If during the course of a transaction a correctable error is detected in either the memory, instruction network, data network, or the mass storage units, then the error is corrected and the transaction is continued without interruption. If an uncorrectable error is detected in one of these units, then it is signalled as a failure in the corresponding data processor. In any case, the error is logged for future analysis.

If in the course of processing a transaction a control processor notices an error through its self-testing circuitry, it simply puts itself offline. This will be noticed by one of the other control processors, which will then take over the error processing of the transaction by attaching to the relevant data processors and aborting and restarting the task according to which of the three phases of operation is being processed, as described above.

Self Testing

A processor may go off-line due to one of three reasons:

1. It may explicitly set itself off-line due to an error that is detected; for example, in the execution of a diagnostic routine.

2. It may be put off-line automatically by the self-checking circuitry detecting a hardware error in the system.

3. It may be put off-line automatically if it fails to execute a sign of life instruction during a period of 10 seconds. During normal operation, the operating system will guarantee the sign of life instruction or execute it at regular intervals.

These three methods for making a processor off-line are designed to detect different types of errors. The totally self-checking circuitry will detect faults which cause false operation of the device. The explicit switching off-line by the maintenance routine is designed to detect latent errors which may or may not cause false operation of the device directly, but would reduce the effectiveness of the self-checking circuitry under the single fault assumption (see section on Self-Testing Property in Totally Self-Checking Circuits). The sign of life mechanism is designed to check software errors that cause the program to be stuck in an "infinite" loop.

When a physical control processor goes off-line it ceases to execute instructions over to control memory and turns off the diagnostic light indicating that it should be replaced. It will also cause a status word in a data memory address space to change, indicating that it has gone off-line. This status word is used to clear any locks that the physical control processor may have left set in data memory. Whenever a lock is blocking another processor it will check the status register associated with the control processor idea associated with the lock to check that it is still on-line. In this way, dangling locks will be deleted whenever they are referenced to protect against failures in the off-line signalling mechanism non-parity as used in the status register.

Initialization

Initialization works as follows. When a processor is powered up, it automatically begins to execute the boot sequence by jumping to a section of control memory that is stored in read-only memory. A boot sequence will the following steps:

1. Determine if the operating system is loaded into control memory by computing a 64 bit check sum;

2. If the operating system is loaded, jump to the diagnostic sequence.

3. If the diagnostic sequence fails, go off-line.

4. If the diagnostic sequence passes, go to the normal scheduling loop of the operating system.

5. If the control memory is not loaded, as evidenced by the 64 bit check sum, then, if the disk is free, load the control memory from the disk.

6. If the disk is not free, go back to step one.

Notice that this load-sequence will work properly both in the case where a single physical control processor is powered up after being replaced, and in the case where the entire CM2 is powered up initially. In the latter case, only one processor will be able to access the disk in the normal interlocking mechanisms. If this process is successful, it will load the disk memory for the others. If this processor is not successful, the others will get a chance due to the normal error detection and sign of life mechanisms (see Error Detection and Correction).